6.828 Fall 2003 Lab 4: System calls, IPC page faults, and fork

Handed out Wednesday, October 15
Part A due Thursday, October 23
Part B due Thursday, October 30
Part C due Thursday, November 6

Version: $Revision: 1.8 $.
Last modified: $Date: 2003/10/28 01:22:50 $

Introduction

In this lab you will implement the basic path through the kernel that handles system calls. To exercise the system call code, you will add a primitive IPC system to allow processes to communicate with each other. Then you will move on to a more sophisticated set of system calls, allowing processes to handle their own page faults. Finally, you will use this page fault interface to implement the Unix `fork` system call from user space.

Getting Started

Download the reference code lab4.tar.gz, located in the labs section for this course, and unpack it into your 6.828 directory as before. You will need to merge our new code for this lab into your source tree.

There have been many changes to the source tree. We list all the changes and then afterward give instructions for merging your current solutions into the tree.

Changes to files you have not edited:

- Added file `inc/syscall.h`, defining the system call numbers.
- Added file `kern/syscall.h` defining the kernel system call handler.
- Added file `kern/syscall.c`, containing implementation of system calls.
- Removed file `lab3.S`, because it is lab 3-specific.
- Modified file `kern/env.h` to add `ENV_CREATE` macro for creating environments from binaries hard-coded in the kernel.
  Changed status to `ENV_NOT_RUNNABLE` for readability.
  Changed names of IPC state variables in `struct Env`.
  Changed prototype for `envid2env`.
- Modified file `kern/init.c` to create various test programs.
- Modified file `GNUmakefile` to define some useful targets.
  Making `kernel.asm` produces a disassembly of the kernel.
  Making `user/foo.asm` produces a disassembly of program `user/foo`.

- Modified file `inc/error.h` to rename `E_IPC_BLOCKED` to `E_IPC_NOT_RECV`.
- Modified file `inc/mmu.h` to add `VPN` macro for readability in later code.
- Modified `kern/Makefrag` to build various user binaries into kernel.
- Added files in the `user/` directory to build user programs.
- Modified `boot/Makefrag` and added `boot/sign.pl` to add a signature to the boot sector. This should allow the boot sector to be used with later versions of Bochs.

Changes to files you have edited:

- Modified file `kern/env.c` to free pages in `env_free`.
  Changed definition of `envid2env` to be consistent with other routines, returning error code directly, and also to add a permission-checking option.
  Changed initialization in `env_alloc` to handle the new IPC fields in `struct Env`.
- Modified file `kern/trap.c` to add `page_fault_handler` and a call to it from inside `trap`.
- Modified file `kern/pmap.c` to do a little more checking in `page_check`. Also added `page_lookup` and `page_decref`. Reimplemented `page_remove` in terms of these. You will want to pick up the new functions, because they are useful for writing some of the code for this lab. If you want our implementation, see the solutions to lab 2 now on the web.
- Modified file `kern/printf.c` to update the error list. (Just copy the new file and add your octal code again.)

To merge your solutions from Lab 3 (and before) into the `lab4/` tree:

- Copy your kern/locore.S, `kern/printf.c`, and `kern/sched.c` into the tree.
- Copy your `kern/trap.c` and then add our new `page_fault_handler` function along with the code in `trap` that calls it.
- Copy your `kern/env.c` and then add our new `envid2env` and `env_free` functions. Also notice that the initialization in `env_alloc` has changed slightly:
- ```
  e->env_ipc_blocked = 0;
  ```
- ```
  e->env_ipc_value = 0;
  ```
- ```
  e->env_ipc_from = 0;
  ```

  becomes

  ```
  e->env_ipc_recving = 0;
  ```

- Copy your solutions from lab 2 into `kern/pmap.c`. Don't just copy the file: you need to make sure you keep the changes to `page_check` as well as the new `page_lookup` function, which you need to write (feel free to look at our lab 2 solution). `Page_lookup` will be useful for completing this lab.

Once you have finished the merge, here are some things to double-check. *Do this before proceeding!* You will save yourself many headaches by getting these correct now, before they cause hard-to-debug crashes.
  ```

- Make sure that `page_check` still passes. There are some new tests in it that check your handling of reference counts. If your handling is wrong, you might put a page on the free list incorrectly.

If you build a kernel and run it after the merge, you should see:

```
6828 decimal is 015254 octal!
Physical memory: 32768K available, base = 640K, extended = 31744K
...
[00000000] new env 00000800
[00000000] new env 00001001
TRAP frame at 0xefbfffbc
             edi  0x0
             esi  0x0
             ebp  0xeebfdfa8
             oesp 0xefbfffdc
             ebx  0x0
             edx  0x800050
             ecx  0x0
             eax  0x1
             es   0x23
             ds   0x23
             trap 0x30
             err  0x0
             eip  0x800b2e
             cs   0x1b
             flag 0x246
             esp  0xeebfdf9c
             ss   0x23
panic at kern/trap.c:143: unhandled trap
```

All the elements in the trap frame should match. The `trap` may be `0xd` (general protection fault) if you have not added the system call trap (0x30) to the GDT yet.

Part A: System calls and IPC

System calls

User processes ask the kernel to do things for them by invoking system calls. When the user process invokes a system call, the processor enters kernel mode, the processor and the kernel cooperate to save the user process's state, the kernel executes appropriate code in order to carry out the system call, and then resumes the user process. The exact details of how the user process gets the kernel's attention and how it specifies which call it wants to execute vary from system to system.

In V6, user processes executed the `sys` instruction to get the kernel's attention. The user process specifies the type of system call with a constant in the instruction itself. The arguments to the system call are also in the instruction stream, or in registers, or on the stack, or some combination of the three. The kernel passes the return value back to the user process in `r0`.

In the x86 kernel, we will use the `int` instruction, which causes a processor interrupt. In particular, we will use `int $0x30` as the system call interrupt. We have defined the constant `T_SYSCALL` to 0x30 for you. You will have to set up the interrupt descriptor to allow user processes to cause that interrupt. Note that interrupt 0x30 cannot be generated by hardware, so there is no ambiguity caused by allowing user code to generate it.

In the x86 kernel, we will pass the system call number and the system call arguments in registers. This way, we don't need to grub around in the user environment's stack or instruction stream. The system call number will go in `%eax`, and the arguments (up to five of them) will go in `%edx`, `%ecx`, `%ebx`, `%edi`, and `%esi`, respectively. The kernel passes the return value back in `%eax`. The assembly code to invoke a system call has been written for you, in `syscall()` in `user/syscall.c`. You should read through it and make sure you understand what is going on.

Implementation

Sanity check

Before you go any further, let's make sure that loading user environments still works.

Look at `kern/init.c`: it creates two user environments (with the `ENV_CREATE` macro) from binaries that have been linked into the program. The binaries in the last lab were hand-written position-independent assembly stored in `lab3.S`. The binaries in this lab are compiled from C sources in the `user/` directory. The program `user/foo` is linked into the kernel in such a way that `ENV_CREATE(user_foo)` calls `env_create` with the right arguments to start it running in a new environment.

We'll start with the user environment `user/idle` (compiled from `user/idle.c`), which is a lot like `spin` from the last lab: it just loops. This may seem like a rather useless program, but in fact it serves an important purpose. When the kernel has nothing else to do, it can run the idle environment, which will keep the processor in user mode so that device interrupts can occur and trap back into the kernel.

Edit `kern/init.c` and comment out the line "`ENV_CREATE(user_hello);`". Now the only environment being created is `user/idle`.

Build the kernel and start Bochs. Set a break point at the loop in `idle`'s `umain`: `vb 0x1b:0x800054`. Then boot the kernel. When you reach the break point, step (`s`) a few times. You should see:

```
<bochs:1> vb 0x1b:0x800054
<bochs:2> c
6828 decimal is 15254 octal!
...
page_check() succeeded!
             Setup timer interrupts via 8259A
             unmasked timer interrupt
```

```
(0) Breakpoint 1, 0x800054 (0x1b:0x800054)
Next at t=9167210
(0) 001b:00800054 (unknown context): ebfe: jmp +#fe
 s
Next at t=9167211
(0) 001b:00800054 (unknown context): ebfe: jmp +#fe
 s
Next at t=9167212
(0) 001b:00800054 (unknown context): ebfe: jmp +#fe
 s
Next at t=9167213
(0) 001b:00800054 (unknown context): ebfe: jmp +#fe
```

(The `t=` numbers will be different.) If you do not see this, do not continue. Go through your merge and figure out what was merged incorrectly. Re-check the list above.

To find the address of `idle`'s `umain`, you can `gmake user/idle.asm` to generate a disassembly.

Change your scheduler (`sched_yield` in `kern/sched.c`) to run the idle environment (`envs[0]`) only when no other environments are runnable: any other environment should take priority over the idle environment.

Rebuild your kernel, repeat the above sequence, and make sure you still get the same results.

System call path

Add a handler for interrupt `T_SYSCALL`. You will have to edit `locore.S` and `kern/trap.c`'s `idt_init()`. You also need to change `trap()` to handle the system call interrupt by calling `syscall` with the appropriate arguments and then arranging for the return value to be passed back to the user process in `%eax`. Finally, you need to implement `syscall` in `kern/syscall.c`. Make sure `syscall` returns `-E_INVAL` if the system call number is invalid. You should read and understand `user/syscall.c` (especially the inline assembly routine) in order to confirm your understanding of the system call interface. You may also find it helpful to read `inc/syscall.h`.

Uncomment the "`ENV_CREATE(user_hello)`". Compile and run your kernel. It should print "`hello, world`" and then panic with a page fault from user mode. If this does not happen, it probably means your system call handler isn't quite right.

User-mode startup

The user programs start running at the top of `user/entry.S`. After some setup, this code calls `libmain()`, in `user/libos.c`. `Libmain` needs to initialize a global pointer `env` to point at this environment's `struct Env` in the `envs[]` array. (`Entry.S` defined `envs` to point at the `UENVS` mapping you set up in lab 2.) Hint: look in `inc/env.h` and use `sys_getenvid`.

`Libmain` calls `umain`, which, in the case of the hello program, is in `user/hello.c`. Note that after printing "`hello, world`", it tries to access `env->env_id`. This is why it faulted earlier. Now that you've initialized `env` properly, it should not fault. If it still faults, you probably haven't mapped the `UENVS` area user-readable (back in lab 1 in `pmap.c`; this is the first time we've actually used the `UENVS` area). Boot your kernel. You should see `user/hello` print "`hello, world`" and then print "`i am environment 00001001`".

Notice that `user/hello` calls `printf`. This is *not* the `printf` from the kernel. (How could it be? There's no `printf` system call!) Instead it is a user-space `printf` implemented in `user/printf.c` and linked into each user program. This `printf` prints into a string buffer and then calls `sys_cputs` on the string. (Recall that the kernel calls `cons_putc` on every character as it is ready to be printed.)

Interprocess communication (IPC)

(Technically this is "inter-environment communication" or "IEC", but everyone else calls it IPC, so we'll use the standard term.)

We've been focusing on the isolation aspects of the operating system, the ways it provides the illusion that each program has a machine all to itself. Another important service of an operating system is to allow programs to communicate with each other when they want to. It can be quite powerful to let programs interact with other programs. The UNIX pipe model is the canonical example.

There are many models for interprocess communication. Even today there are still debates about which models are better for various reasons. We won't get into that debate. Instead, we'll implement a simple IPC mechanism and then try it out.

Implementation

You will implement a simple interprocess communication mechanism using the system call interface you just constructed. It will allow environments to send integer values to other environments. You will implement two system calls, `sys_ipc_recv` and `sys_ipc_can_send`. Then you will implement two library wrappers `ipc_recv` and `ipc_send`.

To receive a value, an environment calls `sys_ipc_recv`, which deschedules the current environment and does not run it again until a value has been received.

To try to send a value, an environment calls `sys_ipc_can_send` with both the receiver's environment id and the value to be sent. If the named environment is actually receiving (it has called `sys_ipc_recv` and not gotten a value yet), then the send delivers the value and returns 0. Otherwise the send returns `-E_IPC_NOT_RECV` to indicate that the target environment is not currently expecting to receive a value.

A library function `ipc_recv` will take care of calling `sys_ipc_recv` and then looking up the information about the received values in the current environment's `struct Env`.

Similarly, a library function `ipc_send` will take care of repeatedly calling `sys_ipc_can_send` until the send succeeds.

System calls

Implement `sys_ipc_recv` and `sys_ipc_can_send` in `kern/syscall.c`. When you call `envid2env` in these routines, you should set the `checkperm` flag to 0, meaning that any environment is allowed to send IPC messages to any other environment, and the kernel does no special permission checking other than verifying that the target envid is valid. Currently `envid2env` just ignores this flag anyway, but later in this lab you will enhance it with permission checking support.

Implement the user-space calls in `user/syscall.c`.

Library wrappers

Implement `ipc_recv` and `ipc_send` in `user/ipc.c`.

Change `kern/init.c` to start *two* copies of `user/pingpong2` instead of `user/hello`. (Note that it's `pingpong2` and *not* `pingpong1` and *not* `pingpong`.)

Boot your kernel. You should see:

```
[00000000] new env 00000800
[00000000] new env 00001001
[00000000] new env 00001802
1802 got 0 from 1001
1001 got 1 from 1802
1802 got 2 from 1001
1001 got 3 from 1802
1802 got 4 from 1001
1001 got 5 from 1802
1802 got 6 from 1001
1001 got 7 from 1802
1802 got 8 from 1001
1001 got 9 from 1802
[00001001] exiting gracefully
[00001001] free env 00001001
1802 got 10 from 1001
[00001802] exiting gracefully
[00001802] free env 00001802
```

Challenge!!! `Ipc_send` is not very fair. Run three copies of `user/fairness` and you will see this. The first two copies are both trying to send to the third copy, but only one of them will ever succeed. Make the IPC fair, so that each copy has approximately equal chance of succeeding.

Challenge!!! Why does `ipc_send` have to loop? Change the system call interface so it doesn't have to. Make sure you can handle multiple environments trying to send to one environment at the same time.

This ends part A. As usual, you can grade your submission with `gmake grade` and hand it in with `gmake handin`. If you are trying to figure out why a particular test case is failing, run `sh grade.sh -x`, which will show you the output of the kernel builds and Bochs runs for each test, until a test fails. When a test fails, the script will stop, and then you can inspect `bochs.out` to see what the kernel actually printed.

Part B: Page fault handling from kernel and user modes

Page faults and memory protection

Memory protection is a crucial feature of an operating system. By using memory protection, the operating system can ensure that bugs in one program cannot corrupt other programs or corrupt the operating system itself.

Typically, operating systems rely on hardware support to implement memory protection. The OS keeps the hardware informed about which virtual addresses are valid and which are not. When a program tries to access an invalid address or one for which it has no permissions, the processor stops the program at the instruction causing the fault and then traps into the kernel with information about the attempted operation. If the fault is fixable, the kernel can fix it and let the program continue running. If the fault is not fixable, then the program cannot continue, since it will never get past the instruction causing the fault.

As an example of a fixable fault, consider an automatically extended stack. In many systems the kernel allocates a single stack page, and then if a program faults accessing pages further down the stack, the kernel will allocate those pages automatically and let the program continue. By doing this, the kernel only allocates the memory that the program is going to use, but the program can work under the illusion that it has an arbitrarily large stack.

System calls present an interesting problem for memory protection. Most system call interfaces let user programs pass pointers to the kernel. These pointers point at user buffers to be read or written. The kernel then dereferences these pointers on behalf of the user while carrying out the system call. There are two problems with this.

First, a page fault in the kernel is taken a lot more seriously than a page fault in a user program. If the kernel page faults, that's usually a kernel bug, and the fault handler will panic. We need a way to remember that the page fault is on behalf of a user program.

Second, the kernel typically has more memory permissions than the user program. The user program might ask the kernel to read from or write to a location in kernel memory that the user program cannot access but that the kernel can. If the kernel is not careful, a buggy or malicious user program can trick the kernel into using its greater privilege in unintended ways, possibly so as to destroy the integrity of the kernel completely. This danger is one instance of a classic security problem known as the "confused deputy" problem. The kernel is acting as a trusted "deputy", which has the special privileges necessary to implement important services needed by untrusted users - but if users can confuse the kernel into using those special privileges in unintended ways, the security model breaks down. (*Challenge*!!! Explore the literature on the confused deputy problem available on the Internet, and identify other aspects of the design of this toy kernel and existing well-known operating systems in which this security risk may occur.)

For both of these reasons the kernel must be careful when handling pointers presented by user programs.

Implementation

In your kernel, you will implement solutions to these two problems.

To address the first problem, you will use a global variable `page_fault_mode` to let the fault handler know when the kernel is manipulating memory on behalf of the user environment. If a fault happens then, the user environment will be destroyed. (Otherwise, if a fault happens, the kernel should panic.)

To address the second problem, you will "sanitize" all user pointers by using `TRUP` ("TRanslate User Pointer"). This macro will leave valid user pointers as is, but will translate all other pointers to `ULIM`, which will definitely cause a page fault when accessed.

Sanity check

Make sure that the fault path from the last lab still works. Change `kern/init.c` to run `user/fault`. Run the kernel. The kernel should panic inside the page fault handler. You should see:

```
[00000000] new env 00000800
[00000000] new env 00001001
TRAP frame at 0xefbfffbc
             edi  0x0
             esi  0x801028
             ebp  0xeebfdfc8
             oesp 0xefbfffdc
             ebx  0x70
             edx  0xeec00074
             ecx  0x0
             eax  0x1
             es   0x23
             ds   0x23
             trap 0xe
```

```
            err  0xfffe
            eip  0x800053
            cs   0x1b
            flag 0x10296
            esp  0xeebfdfc8
            ss   0x23
panic at kern/trap.c:174: page fault
```

You may see `6` for `err`, and slightly different values for `esi` and `ebx`. That's okay. All the other fields should match identically.

Page fault mode

Change `kern/trap.c`'s page fault handler. If a page fault happens while in kernel mode, check the setting of `page_fault_mode` and act accordingly. `Inc/mmu.h` explains the possible page fault modes. If you destroy the current environment, print a message explaining the fault in the format:

```
printf("[%08x] PFM_KILL va %08x ip %08x\n", curenv->env_id, va, tf->tf_eip);
```

Hint: to determine whether a fault happened in user mode or in kernel mode, check the low bits of the `tf_cs`.

Change `kern/syscall.c` to set the page fault mode correctly when handling the user pointer in `sys_cputs`. Make sure you reset the page fault mode when the code finishes handling the user pointer.

Change `kern/init.c` to run `user/buggyhello` instead of `user/hello`. Compile your kernel and boot it. The environment should be destroyed, and the kernel should not panic. You should see:

```
[00000000] new env 00000800
[00000000] new env 00001001
[00001001] PFM_KILL va 00000001 ip f010263d
[00001001] free env 00001001
```

(Your `ip` may be different but should begin `f01`.)

TRUP

The check you added protects against buggy environments that pass invalid pointers, but does not protect against evil environments that pass pointers to valid kernel memory. `User/evilhello` is one such program.

Change the definition of `sys_cputs` to protect itself against malicious user environments by using `TRUP`.

Change `kern/init.c` to run `user/evilhello`. Compile your kernel and boot it. The environment should be destroyed, and the kernel should not panic. You should see:

```
[00000000] new env 00000800
[00000000] new env 00001001
[00001001] PFM_KILL va ef800000 ip f010263d
[00001001] free env 00001001
```

(Your `ip` may be different but should begin `f01`.)

User-level page fault handling

Most page faults encountered during normal program execution are fixable, like the stack example we saw earlier. A typical kernel must track, for each environment, the various regions of memory that are mapped and what to do when faults happen in them. For example, we saw that a fault in the stack region will typically map in a new page. A fault in the BSS region will typically map in a new page and also make sure it is zeroed. In systems with demand-paged executables, a fault in the text region will read the corresponding page of the binary off of disk and then map it in.

This is a lot of information for the kernel to track and get right. Instead, we will push this functionality into user space, where bugs are less damaging. This has the added benefit of allowing programs great flexibility in defining their memory regions. As we will see, this allows so much flexibility that we can implement `fork()` as a user space library routine.

Implementation

We will provide this functionality by allowing user environments to handle their own page faults. During standard execution, a user program will run on the user stack (its top located at virtual address `USTACKTOP`). When a page fault occurs in user mode, the kernel will restart the user environment running the user-level page fault handler on a different stack, the user exception stack (its top at `UXSTACKTOP`). The page fault handler can use system calls to map new pages in order to fix the fault. Then the page fault handler will return, via an assembly language stub, to the faulting code on the original stack. We will call the state of the user environment at the time of the fault the *trap-time* state.

Memory allocation

Each user environment will need to allocate memory for its exception stack. To allow this, implement the `sys_mem_alloc` system call in `kern/syscall.c` and `user/syscall.c`. Note that you will have to choose the error codes to return for the error checking you implement.

Page fault handler

You will add a new system call `sys_set_pgfault_handler()` that allows an environment to register a page fault handler and a stack. The call "`sys_set_pgfault_handler(envid, handler, stacktop)`" will change the page fault handler for the environment `envid`. When a page fault occurs in user mode for the target

environment, the kernel will call the user environment's `handler` running on the stack whose top is `stacktop`.

For this and all system calls that use environment ids, we will adopt the convention that id 0 means "the current environment". This convention is implemented by `envid2env`. Implement `sys_set_pgfault_handler` in `kern/syscall.c`. When you call `envid2env`, set `checkperm` to 1, so that an environment can only change the page fault handler for itself or for a child.

Write the `checkperm` case in `kern/env.c`'s `envid2env` to implement the permission checking just described. Implement `sys_set_pgfault_handler` in `user/syscall.c`.

Change `kern/trap.c`'s `pgfault` to handle faults from user mode as follows.

If there is no page fault handler registered, the user environment should be destroyed with a message like before, except say "`user fault`" instead of "`PFM_KILL`".

Otherwise, set up a trap frame on the exception stack that looks like this:

```
            <-- env_xstacktop
empty
empty
empty
empty
empty
tf->tf_eip
tf->tf_eflags
tf->tf_esp
tf->tf_err
fault_va    <-- %esp when handler is run
```

and then arranging for the user environment to resume execution with the page fault handler running on that stack (you must figure out how to make this happen). Each `empty` line in the frame above is simply a 32-bit word-size space on the exception stack that the kernel does not initialize, but the fault handler in the user environment can use. The `fault_va` is the virtual address at which the page fault occurred.

Remember that `env_xstacktop` is a pointer given to the kernel by a user program. Treat it with the appropriate caution.

If `tf->tf_esp` is already on the user exception stack, then the page fault handler itself has faulted. In this case, you should start the new stack frame just under the current `tf->tf_esp` rather than under `env_xstacktop`:

```
            <-- tf->tf_esp
empty
empty
empty
empty
empty
```

```
tf->tf_eip
tf->tf_eflags
tf->tf_esp
tf->tf_err
fault_va    <-- %esp when handler is run
```

To test whether `tf->tf_esp` is already on the user exception stack, check whether it is in the range `[env->env_xstacktop-BY2PG, env->env_xstacktop-1]`.

You should not need to sanity check the fault handler `%eip`: let the memory protection hardware do that for you.

Three of the empty words are for the assembly routine to use to save the caller-save registers before moving on to C code. The other two are also important for the recursive fault case. If you get stuck, draw a stack diagram for the recursive case and look at how each word gets used.

Next, you need to implement the assembly routine that will take care of calling the C page fault handler and resume execution at the original faulting instruction. This assembly routine is the handler that will be registered with the kernel.

Implement `__asm_pgfault_handler` in `user/entry.S`. There is a commented outline there to help you along. You may find it useful to reread the description from the beginning of this section as well.

Finally, you need to implement the C user library side of all this. Finish `set_pgfault_handler` in `user/pgfault.c`.

Testing

Change `kern/init.c` to run `user/fault`. Build your kernel and run it. You should see:

```
[00000000] new env 00000800
[00000000] new env 00001001
[00001001] user fault va 00000000 ip 0080008b
[00001001] free env 00001001
```

Change `kern/init.c` to run `user/faultdie`. Build your kernel and run it. You should see:

```
[00000000] new env 00000800
[00000000] new env 00001001
i faulted at va deadbeef, err 6
[00001001] exiting gracefully
[00001001] free env 00001001
```

Change `kern/init.c` to run `user/faultalloc`. Build your kernel and run it. You should see:

```
[00000000] new env 00000800
[00000000] new env 00001001
fault deadbeef
this string was faulted in at deadbeef
fault cafebffe
fault cafec000
this string was faulted in at cafebffe
[00001001] exiting gracefully
[00001001] free env 00001001
```

If you see only the first "this string" line, it means you are not handling recursive page faults properly.

Change `kern/init.c` to run `user/faultallocbad`. Build your kernel and run it. You should see:

```
[00000000] new env 00000800
[00000000] new env 00001001
[00001001] PFM_KILL va deadbeef ip f010263d
[00001001] free env 00001001
```

(Your ip may differ from ours but should begin `f01`.)

Make sure you understand why `user/faultalloc` and `user/faultallocbad` behave differently.

This ends part B. As usual, you can grade your submission with `gmake grade` and hand it in with `gmake handin`.

Part C: User-level fork

As the final piece of this lab, you will implement the UNIX `fork` system call as a user space library routine, using the system calls you implemented in part B along with a couple new ones.

Fork

As its process creation primitive, UNIX provides the `fork` system call. Fork copies the calling process (the parent) to create a new process (the child). The only differences between the two observable from user space are their process IDs and parent process IDs (as returned by `getpid` and `getppid`). In the parent, `fork` returns the child's process ID. In the child, `fork` returns 0. The two processes do not share any memory: writes to one process's memory do not appear in the other and vice versa.

V6 UNIX implemented `fork` by copying the parent's segments into new memory for the child. This was, by far, the most expensive part of the call. Since `fork` is so often followed by `exec` in the child (for example, in the shell), these copies are often wasted.

Later versions of UNIX took advantage of virtual memory hardware to allow the parent and child to share the memory until one of them modified it. To do this, the kernel marked the now-shared pages not writable. When one of the two processes tried to write to the page, it would fault. At this point, the kernel would realize that the page was really "copy-on-write" and make a private copy for the faulting process. In this implementation, pages aren't copied until they were written to. This makes `fork` followed by `exec` in the child much cheaper: the child would probably only copy one page (its current stack page) before calling `exec`.

You'll implement a copy-on-write `fork` now, but entirely from user space. This has the benefit that the kernel support is much simpler and thus more likely to be correct. It also lets programs define their own semantics for `fork`. A program that wanted a slightly different implementation (for example, the expensive always-copy version or one in which the parent and child shared memory afterward) can easily provide its own.

Implementation

You will add a new system call `sys_env_alloc` which creates a new environment with an almost blank slate: no address space, no page fault handler, and not runnable. The new environment will have the same register state as the parent environment at the time of the call. The only difference is that in the parent, `sys_env_alloc` will return the id of the newly created environment, but in the child it will return 0. (The child will be marked as not runnable, so `sys_env_alloc` will not return in the child until the parent has explicitly allowed this by marking the child runnable.)

Then you will add system calls to manipulate this blank slate: `sys_map_mem` and `sys_unmap_mem` allow you to modify the virtual address space, and `sys_set_pgfault_handler` from part B manipulates the page fault handlers.

Using these primitives you will build `fork` itself. `Fork` will create a new environment, copy the state from the current environment into the new environment, and then set it running.

System calls

Implement `sys_env_alloc`, `sys_mem_map`, `sys_mem_unmap`, and `sys_set_env_status` in `kern/syscall.c` and `user/syscall.c`.

Change `kern/init.c` to run `user/pingpong1` (start just one instance of it). Build and boot the kernel. You should see:

```
[00000000] new env 00000800
```

```
[00000000] new env 00001001
[00001001] new env 00001802
send 0 from 1001 to 1802
1802 got 0 from 1001
1001 got 1 from 1802
1802 got 2 from 1001
1001 got 3 from 1802
1802 got 4 from 1001
1001 got 5 from 1802
1802 got 6 from 1001
1001 got 7 from 1802
1802 got 8 from 1001
1001 got 9 from 1802
[00001001] exiting gracefully
[00001001] free env 00001001
1802 got 10 from 1001
[00001802] exiting gracefully
[00001802] free env 00001802
```

Fork

Implement `fork` and `pgfault` in `user/fork.c`. The control flow for `fork` is as follows:

1. The parent installs `pgfault` using `set_pgfault_handler`.
2. The parent calls `sys_env_alloc()` to allocate a child environment.
3. For each writable or copy-on-write page in its address space below UTOP, the parent maps the page copy-on-write into the address space of the child and then remaps the page copy-on-write in its own address space.

   The exception stack is *not* remapped this way. Instead you need to allocate a fresh page in the child for the exception stack. Since the page fault handler will be doing the actual copying and the page fault handler runs on the exception stack, the exception stack cannot be made copy-on-write: who would copy it?

4. The parent sets the user page fault handler for the child to look like its own.
5. The child is now ready to run, so the parent marks it runnable.

After the fork, both processes will fault while trying to execute their code. Here's the control flow for the user page fault handler:

1. Kernel propagates page fault to `asm_pgfault_handler`, which calls `pgfault_handler`.
2. `Pgfault_handler` checks that the fault is a write (check `FEC_WR`) and that the pte for the page is marked `PTE_COW`. If not, panic.
3. `Pgfault_handler` allocates a new page mapped at a temporary location and copies the current page contents into it. Then it maps the new page at the appropriate address.

Change `kern/init.c` to run `user/pingpong` (*not* `pingpong1` and not `pingpong2`). You should see the same output you saw for `pingpong1`.

Change `kern/init.c` to run `user/primes`. You should see:

```
[00000000] new env 00000800
[00000000] new env 00001001
[00001001] new env 00001802
2 [00001802] new env 00002003
3 [00002003] new env 00002804
5 [00002804] new env 00003005
7 [00003005] new env 00003806
11 [00003806] new env 00004007
...
```

You might find it interesting to read `user/primes.c` to see all the forking and IPC going on behind the scenes.

Challenge!!! Implement a shared-memory `fork` called `sfork`. This version should have the parent and child sharing all their memory pages (writes in one environment appear in the other) except for pages in the stack area, which should be treated in the usual copy-on-write manner. Use it to run `user/pingpongs`. You will have to find a new way to provide the functionality of the global `env` pointer.

Challenge!!! The prime sieve is only one neat use of message passing between a large number of concurrent programs. Read C. A. R. Hoare, ``Communicating Sequential Processes," *Communications of the ACM* 21(8) (August 1978), 666-667, and implement the matrix multiplication example.

Challenge!!! Probably the most impressive example of the power of message passing is Doug McIlroy's power series calculator, described in M. Douglas McIlroy, ``Squinting at Power Series," *Software--Practice and Experience*, 20(7) (July 1990), 661-683. Implement his power series calculator and compute the power series for *sin*(1+*x*^2).

Challenge!!! Your implementation of `fork` makes a huge number of system calls. On the x86, switching into the kernel has non-trivial cost. Augment the system call interface so that it is possible to send a batch of system calls at once. Then change `fork` to use this interface. How much faster is your new `fork`?

This ends part C. As usual, you can grade your submission with `gmake grade` and hand it in with `gmake handin`.